Lab 8b - Arduino

October 11, 2018

1 ME 30 Lab - Arduino Week 1

Description and Summary: You *must* have performed the pre-lab entitled *FirmataPlusMrYsLab-setup* before attempting this lab, as both Anaconda and Arduino board preparation are required prerequesites. It assumes that you attended the lecture (and/or have access to the lecture slide-set) for some of the prerequesite information.

The primary intent of ME30 is to work on your software programming skills, and we've made a conscious decision not to subject you to too much electronics in this course, even if you already have taken EE98 and gotten some of the basics. There will be plenty of time for that in ME106!

Using some interfaces that we've developed, to you the hardware is going to simply look like software objects (variables) that just happen to do hardware-y things as a consequence of using them via function calls. We'll also show you the "hard" way of doing it, just so that you can see the difference.

You'll also get the bare minimum of hardware information to be able to use the components that are on the boards contained in your hardware kit.

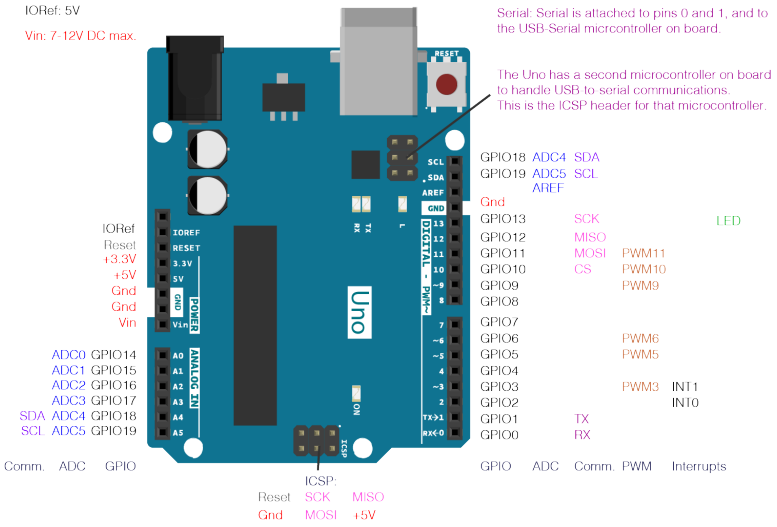
By the end of this lab you should be able to write programs that allow the PC and Arduino to communicate information bi-directionally, essentially using the Arduino as an extension of the PC for interacting with the physical world. ***

1.1 The bare-minimum hardware background (input, output, digital, analog)

The Arduino Uno (the bottom board in your kit stack) is itself a computer with its own program space and local memory for running programs. It's powerful enough to perform many, many control functions that don't require large programs or memory for data handling. Fairly few students require more than their Arduino Uno for their ME106 final projects -- it's usually more than capable unless you're fairly wasteful using its resources.

Above is an image of the Arduino Uno showing the naming of the pins. Unfortunately the image is rotated 90 clockwise, so the "top" edge of the board is shown here on the right. So, flop your head to the right to see it correctly... :-(. This is the board that you can't really see that's on the bottom of the stack. However it's more indicative of what you'll see on the Internet where people are demonstrating their projects with random wires sticking into their board.

Both the Uno board and the YouKnow board that plugs on top of it have two pairs of female headers, one each along the top and bottom edge. The bottom left 8-connector header is where most of the power-related signal congregate, and you don't need to know anything about them. The four leftmost pins on the 10-pin header are mostly redundant and uninteresting to us. All of the voltages directly connected to this board must be in the range of zero to five volts (0-5V) -- so when we say "high" voltage, we're still only talking about 5V, maximum. Nothing to make a



Arduino pins, numbered

grilled cheese sandwich with directly. However this is exactly something that you can do when you get to ME106 with a small amount of extra circuitry that you will learn to design.

The remaining twenty (20) electrical connections are I/O (input/output) pins for attaching to signals, and these are where all of the action is. All twenty pins can be configured to used as either digital inputs or digital outputs. Digital in this context means exactly one of two (binary) states per pin (single electrical connection). You can either sense whether something is attached to them with a high or low voltage, ,in which case they're acting as inputs. An example of a digital input is a light switch -- two positions, one denoting that the light should be on, and another for off.

In the output case, any of these pins can be made to be high or low voltage, signaling hardware connected to them to be in one state or another (that is, acting as outputs). The lightbulb or LED would be an example of a digital output -- on or off. Dimming is a function that we'll deal with in a future week, and is possible using some digital trickery.

The 6-pin connector in the bottom-right corner of the board have the additional capability of being able to sense any input voltage between 0V and 5V, not just high vs. low in the digital sensing case. This ability is important for sensing all of the shades of gray that are present in the real world -- light levels, volume levels, temperature levels, etc.

Each of these twenty I/O pins has a number. When all are used as digital inputs, they are numbered 0-19 starting at the upper-right corner and increasing counter-clockwise -- skipping that 8-pin power header and those other four pins that were mentioned above. When using the pins in the six-pin header as analog inputs, they're numbered separately 0-5, left-to-right. If you look closely enough, most of these pin numbers are labeled on the board.

All of these goodies on the YouKnow (the top) board are soldered in place to one specific pin, respectively. You just need to know which devices are refered to by which pin number. A table with these associations is in `PinMap4.5-4.7.pdf`. These will all be described in more detail later.

1.2 Overview of how Python and the Arduino Uno work together

We're not using the Arduino in exactly the same way that 98%+ of other Arduino users are. Most users use an Arduino variation of the C/C++ programming language (and its own set of software tools) to generate a program file that is loaded over the USB cable into the Arduino for direct

execution.

This differs from how we are using it for this course. You will be writing Python in JupyterLab as you have been for the past few weeks and your program will essentially be remote-controlling the Arduino over the USB cable. This entire time the Arduino is running a single program that knows how to interpret these commands that it receives from the PC, that are generated by your Python program.

The short answers as to why we're using the Arduino this way and not using C/C++ as most users do (and as ME30/ME106 students have been doing for the last 7-8 years) is that this particular solution (combination) is easier for most tasks and allows students to be more productive. It is also the least expensive, most stable and capable, and easiest-to-use setup using Python that exists at the moment. However, this landscape is rapidly changing and soon we anticipate that there will be a hardware solution that is capable of running Python natively (without requiring a PC to drive it), at a reasonable cost that is easy (enough) to use.

The only real downside with this solution is that the Arduino is only able to perfor the set of actions that its interpreting program has been pre-programmed to know about -- and can fit, due to the limited size of its memory. Fortunately its current implementation still allows us to use it for more than enough for this course, as well for many types of ME106 projects, if/when that time comes.

Since your kit includes a standard Arduino-compatible board, if you choose to hop on the standard (C/C++) Arduino bandwagon after you're done with this course, you'll already have all of the hardware to do so. And, we're guessing, you may also appreciate why we're moving to Python for the Mechatronics track from C/C++. If you do, we encourage you to come back and let us know in a year or two about your experiences and feelings on this matter.

1.3 Introduction to the course hardware kit

The hardware kit to we are using for this course is a standard Arduino Uno (clone), a custom-designed-for-SJSU-Mechatronics stacked-on extension board, and a USB cable.

The only thing to say about the particular brand of Arduino Uno-compatible board that we are using is it's clone rather than a "genuine" Arduino board, and is less than half the cost of the former. Clone Arduino boards probably constitute 80% of the Arduino-compatible boards out there, so there's nothing evil or unusual about Arduino clones. The Arduino board design is freely-available to anyone that wants to make similar boards and the only restriction is that they're not called "Arduino" boards, but rather must be called Arduino-compatible boards for legal and branding reasons.

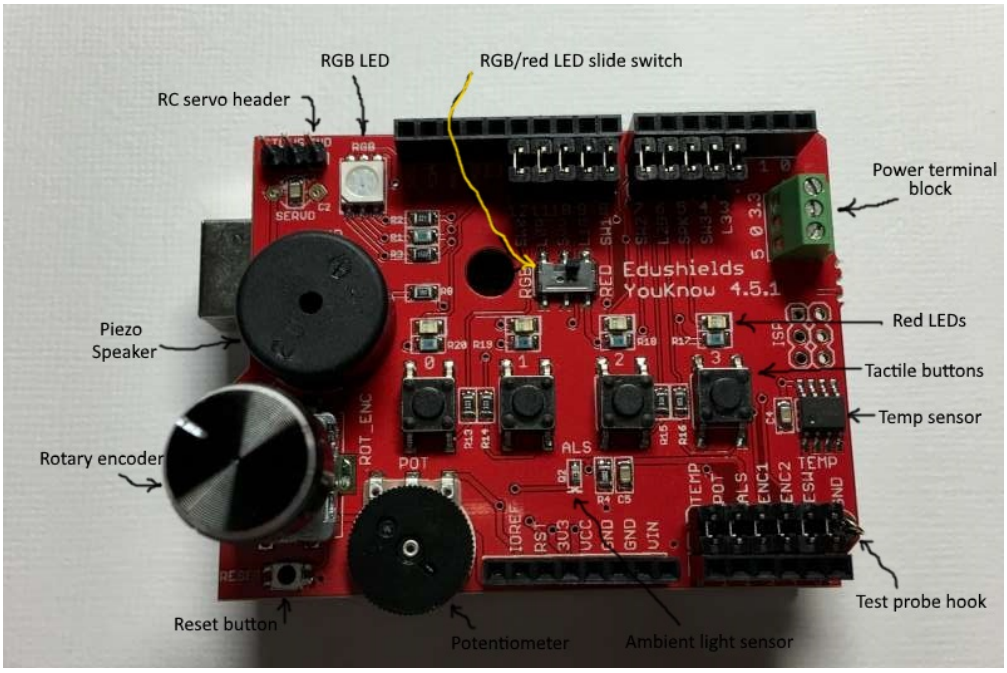
The custom extension board (the EduShields YouKnow) contains a number of sensors and indicators. Being already assembled, it's a much more convenient and stable setup than having a wad of wires and boards connected together for students at this point in the program. However this board can simplify post-ME30 projects (or prototypes) by simplifying them because of what's already built-in. Most students use one or both of their boards as part of their later ME106 and ME190 coursework.

Generically, any board that connects on top of an Arduino board (of this shape and size) is called a *shield*. You may see this term show up in the Arduino literature.

The final piece of the kit is a USB cable to join the PC to the "Arduino" board. This cable enables both the communication to and provides the power for the board stack.

The EduShields YouKnow board contains the following program-accessible components.

Program-accessible devices:



YouKnow 4.x board photo, annotated.

- Four momentary, normally open, tactile switches/pushbuttons
- Four red LEDs (light-emitting diodes)
- One RGB (red-green-blue) LED
- One piezo-electric speaker for tone generation
- One rotary encoder (the tuning knob)
- One rotary dial potentiometer (continuously-variable resistor)
- One ALS (ambient light sensor)
- One 10mV/degF (LM34) temperature sensor
- One three-pin servo motor header

Other support hardware:

- One reset button
- One three-screw terminal block for powering off-board devices
- One test probe GND (ground) hook

For more detail about the YouKnow board, refer to the *EduShields YouKnow v4.X Feature Summary* document, available from the board designer (also possibly from the course website).

1.4 Exercise 1: Characterize those devices!

For each item in the "Program-accessible devices" list above, specify if the component is an input or an output, and whether it's digital or analog.

1.4.1 Features of PyMata and FirmataPlusMrYsLab

Firmata is the name of the remote-control software that has been used to communicate between a host PC and an Arduino board for as long as the Arduino has been around (a little over a decade now). We're using an enhanced version of this software that includes the ability to use a common servo, two-channel rotary encoder capture, and a few other devices that we're not usings. The additions were made to support being able to control a little, wheeled bot, which the "standard" version of Firmata didn't support. Since we really like servo control and rotary encoders, we're using his particular set of extensions. "Mr Y" is Alan Yorinks, whose persona-in-code can be found here at Github.

PyMata is the name given to using this extended Firmata protocol specifically using Python.

This version of software supports the following features that we'll be using throughout the remainder of this course. There are other features that aren't specifically mentioned here, but they aren't a secret and can be found at the previously mentioned Github URL above.

- General-purpose digital input sensing
- General-purpose digital output control
- Analog input sensing
- Pseudo-analog (*PWM*, or *pulse-width modulation*) output control
- Servo control
- Rotary encoder input

1.5 So how do I talk to the Arduino from Python on the PC?

Just like software of any significant complexity, there are multiple layers of software that you can put on top of the problem that give you easy (or *easier*) access to the hardware. Just as you *could* hand-crank the engine to start it, and pull on throttle and brake cables to propel and slow an (older) car, your interface is a higher-level one -- namely an ignition key, and gas and brake pedals. FirmataPlusMyYsLab provides a software interface that's very similar to the one that is used by the C/C++/Arduino language that is typically used. We'll first demonstrate that one, then what we've done to put another layer of software on top of that to simplify coding even more.

The most common basic thing that one can do with most pins on a microcontroller is to either change their state to high or low voltage levels when they're outputs, or to test them being high or low when they're inputs. Using pins this way is called GPIO, or general-purpose I/O.

The first task is to specify whether a given pin is a digital input or a digital output. This is done in both Arduino/C and PyMata by calling a function named something like `pinMode(pin_number, mode)` with the pin number you are dealing with (numbered 0-19 on the Uno), and (basically) either INPUT or OUTPUT, appropriately. Note that although you can change a pin back-and-forth between input or output, one typically only sets it one way at the beginning of the program and leaves it in that mode.

If you are sensing a voltage level on that pin, you specify *input* mode and then, as often as you need to, you *read* the voltage level with a function named something like `digitalRead(pin_number)` to find out if the voltage level is digital high or low. Similarly for output, you specify *output* mode then call a function like `digitalWrite(pin_number, level)` to change the pin to the desired low or high level. For the *most* part, it's basically that easy. With input sensing you can determine if a button is pressed, and with output setting you can turn on or off an LED.

With those functions, combined with a function that allows you to wait for a specified number of milliseconds to *delay* (or *"wait"*, or *"sleep"*), one can blink an LED connected to a given pin once per second with code *similar* to the following:

```
# Example 1

# Blink LED connected to pin 5 every second
pin_mode(led_pin, OUTPUT)

while True:
    digital_write(led_pin, HIGH)
    delay(1000)    # 1000 milliseconds = one second
    digital_write(led_pin, LOW)
    delay(1000)
```

Or similarly, to turn *off* an LED when a push-button is pressed:

```
# Example 2

# Turn on light as long as push-button is held-down (or latching-button is down)
pin_mode(button_pin, INPUT_PULLUP)
pin_mode(led_pin, OUTPUT)

while True:
    if digital_read(button_pin) == LOW:
        digital_write(led_pin, HIGH)
    else:
        digital_write(led_pin, LOW)
```

One thing that you *may* have considered in the first (LED blinking) example above is that you don't really know if it's the case that the LED is designed to turn on when the voltage is high or when it's low -- but that flipping between the two is "sure" to get it to blink, right?

You'll learn in ME106 that circuits can be designed so that an LED can turn on if the voltage is either high or low, and that a button can be press-detected with either a high or low voltage output. Clearly in the second example above, the sense of the LED being "on" is the opposite of that of the pressed-button.

The point of all this is that depending on the LED or button circuit design, your code has to deal with all of these possibilities, and would have to change substantially if changing from one design to another for either LED or button types. However, a simpler *interface* can simplify this problem greatly, and that's what we've done for you.

In each Python program you write, you will have to import either all of *youknow.py*, or just the symbols (variables, etc) that you need from it. Depending on how you import them, you either refer to these symbols with just their name, or you have to prefix them with "`youknow.`".

1.5.1 Exercise 0: youknow.py is your API guy

You should have already installed a copy of `youknow.py` on your machine in a location that JupyterLab will always find it when you ask to `import` it. A redundant copy of this file is also in your notebook directory, but called `youknow.txt`. Open this file in your favorite text file edito, or JupyterLab.

The only contents that you need to be aware of at the moment is the section entitled *Digital pin assignments*. Each one of these constants are set to the Arduino digital pin number for each device on the board, and here for you to use. It's the Python-equivalent to the table that you saw on the PinMap PDF document agove.

Note `PIN_SW0`, `PIN_SW1`, `PIN_SW2` and `PIN_SW3`. The `SW` prefix stands for *switch*, which is what EEs call the class of devices that include push-buttons like ours. The other four symbols you'll need are `PIN_LED0`, `PIN_LED1`, `PIN_LED2` and `PIN_LED3`.

The entire API (application programming interface) to everything on this board to allow you to use it easily is contained in this file.

1.6 LEDs and buttons as Python data types

For each device on the YouKnow board, we've created a separate data type just for you!) for each class of component, along with their companion functions. You may have seen examples of these types of variables, called *objects*, and the functions that are attached to them, called *methods*. If you have used either the String or List data types before this, you have seen this syntax before.

If you're using one of Python's built-in class/object data types (`int`, `float`, `str`, `list`), *instances* (or *objects*) of these types are automagically identified and created for you the first time that you refer to them. You have seen already that referring to a variable before it is set will give you an error, which should make it clear that these objects (variables) are created the first time that their values are set.

However, with instances of types that you (or module creators) define, you have to *declare* them by specifing the function that creates such a variable. In some cases, like for lists, you can do either -- set them first to a value (list1 below), or to declare them (list2 below), as shown below:

```
list1 = [1,2,3]
list2 = list((1,2,3))
```

So, an example of creating `LED` and `Button` objects to represent the hardware on the board, connected to Arduino pin 11 and 12 respectively, and is (basically) as follows:

```
# Example 3

# Turn on light as long as push-button is held-down (or latching-button is down)
led    = Led(board, 11, activeLevel=1)
button = Button(board, 12, activeLevel=0)

while true:
    if button.isPressed():
        led.on()
    else
        led.off()
```

Now compare this code (Example 3) to the code in Example 2. Notice that with a good set of interfaces (or just functions, if that's what you restrict yourself to), it's possible to think about problems at a much higher and simpler level, rather than having to flog all of the details in every line of code. The thought process is often just stating what you want at a high-level, and then coming up with data and functions that implement those tasks in a top-down fashion.

1.6.1 Exercise 1: Enough already... let's talk to the Arduino!

Assuming that your software for using PyMata is installed and working, you should have your board(s) connected via the USB cable.

In a code cell in your turn-in notebook, copy-and-paste the following code and run it. Describe the results.

```
#
# Blink all five LEDs on the Uno+YouKnow (the Uno's built-in
#   LED on pin 13, and the YouKnow's PIN_LED[0..3])
#
from pymata_aio.pymata3 import PyMata3
from youknow            import Led, PIN_LED3

if __name__ == "__main__":
    board = PyMata3()
    led = Led(board, PIN_LED3)

    while True:
        led.on()
        board.sleep(0.5)
        led.off()
        board.sleep(0.5)
```

There are two snippets of code here that may not be obvious. First is the `if __name__ == "__main__":` business. Depending on how your code is put together, it is sometimes difficult for Python to know where your program is supposed to start. This statement ensures that Python starts executing at the right time and place.

The second new piece is that Pymata needs to perform some of its own initialization before you start using it, and creating a `PyMata3` starts this process (one piece of which is finding which USB port has the Arduino connected to it). This object must then be passed to every PyMata-related function (directly or indirectly) so that it knows how to talk to the board again. You'll learn more about this mechanism towards the end of the course when we show how objects actually work. At this point in time, just know that every API in which an `Led`, `Button`, `Speaker`, etc object is created, that you'll need to pass in this `board` object that you created early in your program.

1.6.2 Exercise 2: The other flavor of `import`

In another code cell, copy-and-paste the same program contents, but use "`import youknow.py`" instead of the "`from youknow ...`" style of import used above. Make whatever changes you need to the rest of your code to use this style of import.

Explain why you might prefer one method of `import` over the other.

- Hint#1: They're both useful, so your answer will have at least two parts!
- Hint#2: The `import` statement was first introducted in Section 3.2 of *Think Python*. Review the syntax of how imported names are used using that method, and the one used above.

1.6.3 Exercise 3: Light those LEDs, discretely

Problem statement: Each button on the YouKnow has a red LED directly above it, with each set numbered 0 through 3. Do the required `import` to get the `PIN_SW<n>` and `PIN_LED<n>` symbols out of *youknow.py* whichever way that you'd like to.

Write a program that lights up its nearby LED only when the pushbutton is pressed. For this exercise, use four different objects (variables) for each of the switches and LEDs. Call them `sw0...sw3` and `led0...led3`. You should not use any higher-level data types like lists, tuples, and dictionaries (that is, any sequence types) or even the `range` function.

Explain how great it would be to write a program this way if you had 100 buttons.

2 "List"ing your concerns

Whenever you have a group of things that are all similar and are operated on as a group, it's very, very natural to hold them in a list, super-especially if there's a natural order or association between them. The LEDs and buttons on the board are two such examples. Not only are they both refered to as 0 through 3 (how convenient!), but there's an association between an LED and a button in the previous exercise, that correlates one with the other based on their index in their list.

It should occur to you that there is at least one way to use one or more `Lists` (or `Tuples`) to help you solve the previous problem. Perhaps you even wanted to use one rather than implement Exercise 3 as you were asked to do. If so, good for you! Given a list of LEDs and a correspondingly ordered list of buttons, you walk the button list looking for which ones have been pressed, and then manipulate each LED accordingly. This approach requires two lists, one for buttons and one for LEDs.

The LED list would look something like: `python leds = [ led0, led1, led2, led3 ]` using the values of `led<n>` that you created before. However, and even more concise way of declaring a list like this is:

```
leds = [ Led(board, PIN_LED0),  Led(board, PIN_LED1), Led(board, PIN_LED2), Led(board, PIN_LED3
```

And (arguably) even easier to read if you arrange it like this:

```
leds = [ Led(board, PIN_LED0),
         Led(board, PIN_LED1),
         Led(board, PIN_LED2),
         Led(board, PIN_LED3) ]
```

2.0.1 Exercise 4: Light those LEDs, smartly

Define lists like that demonstrated above, one for the LEDs and one for the buttons. Walk through each of these lists, implementing the same behavior as that in Exercise 3. Also, do not hard-code the length of the list anywhere in your code, but derive it from the (spoiler alert!) length of the list. Coding like this has the advantage that if the number of items in your

One of the three analog input devices on the board is the rotary potentiometer. It is the black dial that extends below the bottom-left edge of the YouKnow board. Physically a potentiometer is a variable resistor. By rotating it you cause it to vary its resistance, and this change of resistance can be sensed by measuring the voltage change using an analog input pin on a microcontroller. The pin to which the "pot" is connected is `youknow.PIN_POT`.

The speaker on the board is a digital output device, even though it is clearly able to generate more than a single tone frequency. There is only one output voltage on this type of device (so the volume cannot be varied), but the frequency at which the alternating digital high and low voltages are produced is what causes the pitch to vary. There is no way to vary the volume of the tone. The pin to which the speaker is connected is `youknow.PIN_SPEAKER`.

The APIs for using the potentiometer and speaker are as follows:

```
pot = Pot(board, pin, analogInBits)
pot_value = pot.fraction()

spkr = Speaker(board, pin)
spkr.play(frequency)
spkr.playStop()
```

For an Arduino Uno board, you should use the value `8` for `analogInBits`. The is a feature dictated by the processor chip used on the Uno.

2.0.2 Exercise 5: Don't panic -- let's be analog-ical about this.

Your task is to have the speaker play a tone that varies with the (angular) position of the potentiometer, but only when one of the four buttons is pressed. You may choose which button you use.

Don't feel like you have to swing for the fence and solve this task in one shot. Build the program up by first dealing with generating a tone. Then vary it based on the value returned by `pot.fraction()`, or when pressing the button. Then work on getting whatever you last need to do to complete the solution. Always resist the urge to write the whole program top-to-bottom and then try to debug the whole thing. It's always easier to debug less new code than it is to debug a lot of it. Trust us!

Note that when you ask the speaker to play, it will play until you call playStop() to shut it up.

Hint: You may be able to get away with being a little lazy in the exercise, because there's no problem asking the speaker to start playing a tone if it's already playing it. So, subsequent calls to `play()` won't really matter, and this will make writing this program easier because you won't have to determine if the button was *just* pressed down, or if it's *still* pressed down.

This is an important point that you need to understand. It isn't always going to be the case with every software model for a device that you can repeatedly "start" a device when it's already running. Have you ever turned the ignition key to start a car that's already running? Yup -- subsequent starts after the initial one aren't pretty.

You will likely have an exercise in the future that depends on your understanding this distinction, and you will learn how to deal with it.